Software Specification by Graph Grammars

M. Nagl*, G. Engels*, R. Gall+, W. Schäfer*

* Angewandte Informatik, University of Osnabrueck
Postfach 4469, D-4500 Osnabrueck, Fed. Rep. of Germany

+ Lehrstuhl fuer Programmiersprachen, University of Erlangen-Nbg.
Martensstr. 3, D-8520 Erlangen, Fed. Rep. of Germany

ABSTRACT

The following paper demonstrates that programmed sequential graph grammars can be used in a systematic proceeding to specify the changes of high level intermediate data structures arising in a programming support environment, in which all tools work in an incremental and syntax-driven mode. In this paper we lay stress upon the way to get the specification rather than on the result of this process. Therefore, we give here some approach to specification engineering using graph grammars. This approach is influenced by the syntactical definition of the underlying programming language or module concept etc. to be supported but also by the idea of the user interface.

1. INTRODUCTION

The software which is to be specified by graph grammars in the following paper is a *programming support environment* (other names: software development system, programmer's workbench, etc.), i.e. a set of tools implemented by software which itself facilitates the development of software. The user of such an environment usually is a programmer. /Bu 80/ summarizes the requirements of an (in this case classic) environment, tailored for the programming language ADA. The idea of such environments is (1) to ease programming, (2) to improve the reliability of software, and thereby (3) minimizing the overall costs for software within the whole software life cycle (problem analysis, design, implementation, validation, integration, installation, maintenance).

The aim of the project *IPSEN* (Incremental Programming Support Environment) which is carried out at the University of Osnabrueck in cooperation with other universities is to develop and implement such an environment. Within IPSEN all technical activities of software development are investigated, which means that support already begins when the design has started, i.e. a part of the specification has been worked out.
IPSEN has the following characteristics

- *incremental mode*: The input is given in terms of language portions (increments) rather than by arbitrary text strings. Analysis, evaluation, or execution is even possible for partial programs, specifications, etc.. This avoids a correction cycle which e.g. for programs consists of reediting, recompiling, relinking before execution after a change.
- *syntax-directed*: Any input is immediately checked, corresponding to context free as well as context sensitive relations and incorrect inputs are rejected. Also, all consequences of the input or a change of an increment are displayed. Therefore, a (partial) program or specification can never be syntactically incorrect. The admissible possibilities for increment inputs or changes are indicated to the programmer by menues or help information.
- *command-driven*: The user specifies by a command what he wants to do rather than only putting in the corresponding text string. Therefore, the system knows the user's intention which eases the analysis for syntactical correctness. On the other hand, parts of the concrete syntax as word symbols and delimiters can be generated automatically by the system.
- *high level intermediate data structures* : Incremental mode enforces that all information contained in an external presentation of a

program, specification, etc. is contained in and can be accessed from an intermediate data structure. On the other hand, support of program development especially means that messages corresponding to syntactical or semantical errors, or reporting on some kind of evaluation or execution are given in terms of constructs of the corresponding programming language and not in those of internal characteristics of the underlying machine. These intermediate data structures are regarded to be graph-like here. Therefore, we call the intermediate code of a program system the system graph, that of a single program module the module graph, and so on. These graphs are the centers of all activities corresponding to system changes, module changes, etc..

- *uniform user interface*: The user interface for all tools is styled uniformly. Thus the user must not realize the change of an activity from one tool to another.
  IPSEN is implemented on a remote mini-computer, which, together with all tools, results in a *programming support machine*.
- *adaptable*: Of course, the chosen module concept or programming language to be supported heavily influence the concept of a software development environment. As a consequence, one major goal of our approach is to think about a design for IPSEN, which is adaptable to other module concepts as well as to other programming languages etc..
- *integrated concept*: Here, integrated means: (1) that most of the activities arising in the software life cycle are supported, (2) that the user interface is uniformly styled, as mentioned above, and (3) that tools are offered which combine related activities which are regarded to belong to different phases of software development in classic environments.

Because of the incremental mode of the environment there is no sequential division of software development activities as suggested by the terms of software life cycle models. For example, the distinction between design, integration, and maintenance of a software system can no longer be sustained. At the time a part of the specification of the software system is put into the system, the (partial) integration can start (check for consistency of intermodular connections) and at the same moment the maintenance can begin (e.g. change due to a variation of the requirement definition). Corresponding to this view we have divided the activities within software development when using an environment like IPSEN into the following three main problem areas:

- *Programming in the Large* (containing the design of the software system using any specification language, transformation into an implementation language, integration, maintenance of the software system etc.),
- *Programming in the Small* (module design, module coding, validation, module maintenance),
- *Organizational Items* (project management, project organization, variant/version control, release control, support of documentation, etc.).

Of course, these three problem areas cannot be strictly separated. For example, one result of Programming in the Large can be a skeleton for each module, where the interface (export, import) of the module is fixed. On the other hand, within Programming in the Small a module can only use those resources, which are imported, and, conversely, all resources have to be realized, which are exported. So, also these problem areas are interleaved. Moreover, integrated tools, as mentioned above, cannot be designed and implemented without having managed an *interaction between the various graph-like data-structures*. If, for example, one implements a tool handling all the tasks which have to be carried out when the export of a module is changed, then this tool must control activities corresponding to (1) project management (as not everybody is allowed to do this change), to (2) project organization (as the cost of this change should be estimated), to (3) release control (as this module now is no longer

accessible), to (4) variant/version control (as the old system which contained the module may further exist as a special variant), to (5) specification within Programming in the Large (as all implications of this change have to be found out and corresponding changes, namely within the import clauses, have to be carried out), to (6) Programming in the Small (as the new import also leads to changes within module implementation), to (7) documentation (as the technical documentation has to be altered also) etc.

In this paper we mainly deal with Programming in the Small and only to a certain extent with Programming in the Large. Furthermore, all aspects of evaluating and executing (partial) program modules or (partial) program systems are not regarded here. So, we concentrate exclusively on *syntactical aspects* here (including the context sensitive syntax). The topic to be considered in detail is, which kind of incremental syntax-aided editing in a broad sense is possible and reasonable within Programming in the Small and Programming in the Large.

In order to describe how modules and programming systems are built up and changed, we use *graph grammars as a specification instrument*. This specification is given for the graph-like intermediate data structures. Here, specification has a two-fold meaning: on one side we make clear how these incremental changes on the module or system graph look like, therefore using the term specification in the sense of making things precise on some more abstract level. On the other hand, we will show that this specification is also a specification in the software engineering sense, i.e. that it yields a detailed guide how to write the software realizing IPSEN. This graph grammar specification uses sequential programmed graph grammars. Here, sequential means that one rewriting step takes place after the other, programmed that such a rewriting step internally consists of a sequence of applications of productions where so-called control procedures determine the order of applications. In this paper no formal details about graph grammars in general and sequential programmed graph grammars in particular are given. The reader is referred to /Na 79/.

For high level intermediate data structures we use *graphs* rather than trees (together with attributes). The reason is that (1) graphs are a uniform model which can be applied to internal high level intermediate codes of all problem areas, (2) there is no (artificial) distinction between the information which can be expressed within trees or outside trees, and (3) also aspects of evaluation and execution of program modules and program systems can be treated by the same model, as further information for evaluation and execution purposes can be integrated without leaving the class of admissible intermediate data structures. This, however, is not a reasoning for avoiding attributes at all. Attributes are necessary for expressing values. We pledge for using the same model for all *structural* information.

This paper is based on /Na 80/ where the concept of an integrated programming support environment is sketched originating from an incremental compiler. Other approaches for programming support environments started from different points of view, as GANDALF /Ha 82/, MENTOR /DG 80/, the Program Synthesizer /TR 81/. Rather independent from each other they all developed similar concepts. However, regarding graphs and not trees as internal structures and using graph gammars as specification instrument is specific to IPSEN. Parts of the presentation of this paper can be found in more detail in /ES 82/ and /GA 82a,b/. A forthcoming paper will discuss the overall concept of IPSEN.

The organization of this paper is as follows: Most of the paper, namely sections 2-6, deal with the problem area Programming in the Small, which is presented in detail. For that, the input mode of syntax-aided editing is given in sections 2-4 which later is generalized to cover the full incremental mode, where inputs, changes, and deletions are put in in any order. We begin with comments on the user interface in section 2, then fixing the shape of the module graph in section 3 and, finally, constructing the programmed graph grammar for specifying the module graph changes in section 4. Then user interface and

programmed graph grammars are revised to handle the full incremental mode in sections 5 and 6, respectively. Section 7 outlines that a completely analogous proceeding can be carried out for the problem area Programming in the Large.

2. COMMENTS ON THE USER INTERFACE

Before starting with incremental editing, it is useful to sketch the *state of the art* how a program nowadays is put into a computer. In most cases the source code of a module is edited by a usual text editor irrespective that is has to be written in some formal notation and it is not arbitrary prose. Then it has to be analyzed syntactically (usually a part of compilation). After a sequence of syntactical corrections the module is syntactically correct. Changing this module means to start again with text editing and reanalyzing. This proceeding is inefficient because of two reasons: (1) it takes a long time to know about syntactical errors, and (2) it is inconvenient to force the programmer to learn unimportant details of his programming language as e.g. the symbols of the concrete syntax.

As the user interface of our syntax-aided editing tool has a deep influence on the graph grammar specification we start our discussion with giving some comments on the user interface in this section.

We suggest a division of the screen into three different areas: (1) the *working area* contains a part of the source code of a module in Programming in the Small, a portion of the specification in Programming in the Large, some fragments of documentation, when supporting the editing of technical documents, etc., (2) the *command area* contains menues for command selection, text fields for parameters corresponding to selected commands etc., and (3) the *status line* reports on the tool which is used, the expected reaction time etc. The latter is no longer regarded in this paper. Working area and command area contain two different cursors indicating the actual position. These are called *working cursor* and *command cursor* in the following text.

Let us regard a *fragment of a session* within which a PASCAL procedure with name EXAMPLE is put in making use of syntax-aided editing (cf. fig. 1). We assume, that the skeleton of this procedure is already displayed in the working area. The working cursor always points to the position where the next input is area. The working cursor always points to the position where the next input is expected. In our case (cf. fig. 1.a) this cursor is located after the procedure head. Within the command area a menue is displayed showing all possible inputs of the user, i.e. in this case all possible declaration alternatives. Now, the user selects the third alternative for a declaration, namely ITD for type declaration.
Within the command area the menue disappears and a frame for type declaration is presented (cf. fig. 1.b). In this frame the word symbol type and the equal sign are already contained. Therefore, the user need not know these symbols of the concrete language syntax. The command cursor is at the position of the type identifier. The comment lines indicate all input fields (by dot sequences) within a frame and make clear, which kind of input is allowed. Now, let us assume that the user types in the string PERSON for the type identifier and then presses a special button "next" indicated by a right arrow. (This next button, of course, can be the corresponding cursor movement key.) As there is only one input field in this frame, the frame is already completed.
The frame, which can be regarded as some cutout of the source code (enriched with detailed information), is now transferred to the working area (cf. fig. 1.c). The working cursor now is at the position of a type definition. The command area immediately shows all possible alternatives. We assume now that the user chooses the alternative for a record type definition.
As the frame consists only of the pair of word symbols record and end in this case it is immediately transferred to the working area, the working cursor being at the position where a record component can be put in (cf. fig 1.d). Now, the user may indicate that he wants to put in a record component.
Within the command area the frame for a record component or a sequence of

components of the same type is displayed (cf. fig. 1.e) which contains only one input field for a list of identifiers. Now, the user inputs the string FIRST_N then presses the next-button by which the separating comma symbol is automatically generated, then inputs the string LAST_N and then, by pressing two times the next-button, indicates that the list of identifiers for record components is completed.
Then, the frame is transferred into the working area (cf. fig. 1.f.) the working cursor being at the position of a type definition. Thus, the displayed menue is the same as in fig. 1.c. Now, the user selects one alternative for type definition and the dialog proceeds anyhow.

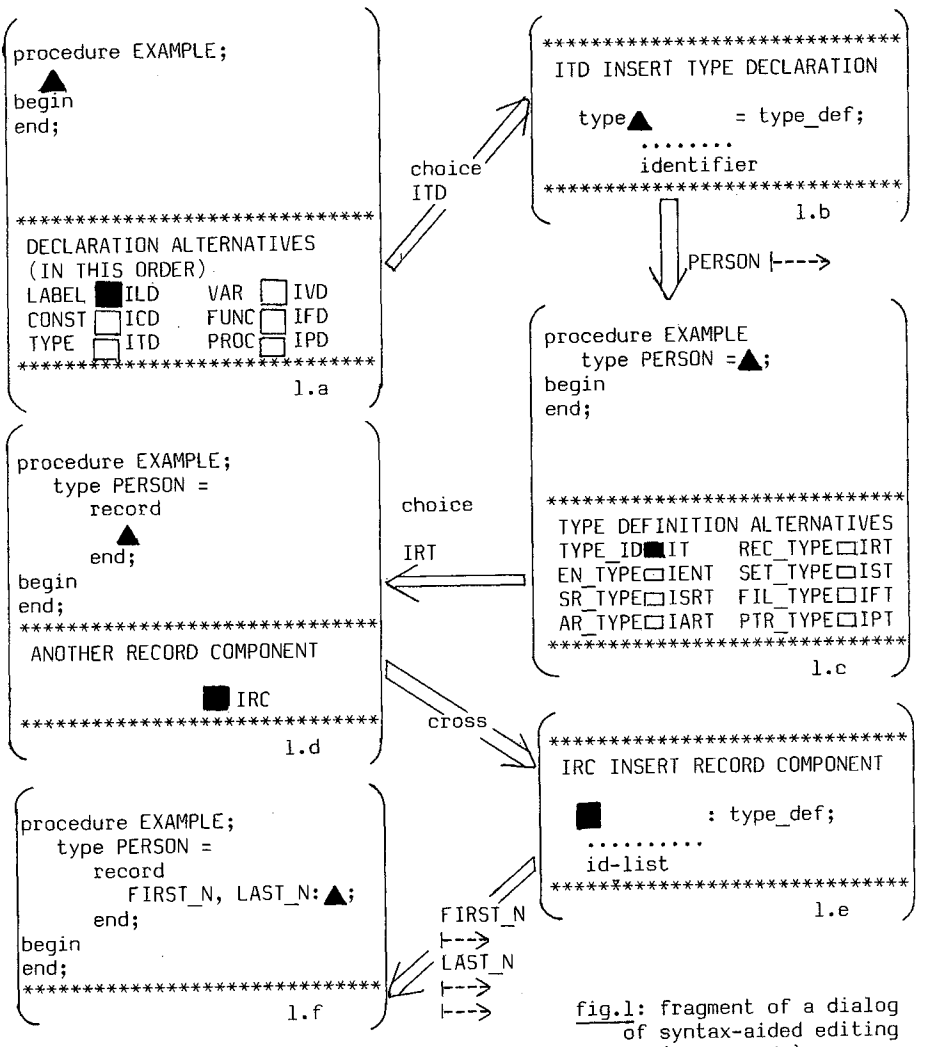


fig.1: fragment of a dialog of syntax-aided editing (input mode)

Above, we have sketched the *menue selection mode* for putting in commands. As menues might be hierarchical, menue selection mode may need more than one menue to be displayed until the actual command can be selected. To abbreviate this, there is also a possibility for putting in a command name by a *text string*. We make use of this mode later. This mode is intended for the more experienced user. In this mode the frames may also have a simpler shape. Furthermore, there is some mechanism to switch between these different command input modes which is not explained in this paper.

What can we learn from the example dialog of fig. 1? The input of a language increment is started by naming an insert command for this increment, which can be done either by selection from a menue or by putting in the corresponding text string. Increments may either be simple, as a type identifier, a record component name etc., or they may be complex, as a type declaration or a type definition. Complex increments are related to structured frames. These frames contain comments to indicate input fields and to give hints what kind of input is expected. All symbols of the concrete syntax are generated. Therefore, the user is liberated from learning the concrete syntax of the underlying programming language.

The *complete syntax* of any input is *immediately checked*: This means (1) that it is immediately checked whether an increment is possible in a special location at all, (2) that the context free syntax rules of the increment (e.g. whether an increment identifier is correctly built up) as well as (3) the context sensitive syntax rules of this increment (e.g. whether a record type declaration does not contain two components with the same name, or whether a variable which is used is also declared) are fulfilled. Therefore, no syntactically incorrect (fragment of a) module is possible (corresponding to the inputs which have already been made).

We have stated above that structured frames correspond to complex increments. However, *frames* and *complex increments* are *not the same*. In fig. 1.b the frame of a type declaration consists only of the concrete syntax symbols type and = and an input field for the type identifier in between. The type definition is of course a part of the type declaration increment but not a part of the frame. The reason is that a type definition may be arbitrarily complex and, therefore, may need arbitrary many lines of source code which cannot be displayed in a region of fixed size on the screen. So, the frame to a complex increment only corresponds to the concrete syntax symbols and simple increments of the uppermost level of the increment. If a complex increment contains further complex increments, then these also have frames corresponding to their uppermost level and so on.

3. THE MODULE GRAPH

The internal high level data structure for a single module, i.e. the *module graph*, is a graph with labelled nodes and edges (cf. fig. 2.). There are different *labelled nodes* for language constructs and different *labelled edges* for expressing different relations between them.
Some nodes correspond to simple increments as the EXAMPLE-node or LAST_N-node. Placeholder nodes express those parts of the module which have to be refined later. In fig. 2 there is a ph_td-node for a type definition which is not filled out, a ph_decl-node for further declarations and a ph_stmt-node for at least one statement in the procedure body. Complex increments correspond to subgraphs and not to nodes. These subgraphs consist of syntax nodes (type-node, proc-node) which indicate the class of increment, simple increment nodes, and placeholder nodes all connected by appropriate edges. Finally the cursor node with its edge uniquely characterizes the actual input position.
Labelled edges express various relations between syntactical constructs. The n-edges express that the target node is the name of a type, procedure, variable etc., the td-edges connect the type or variable identifier with the corresponding type definition, the c-edges point to the components of a record or enumeration type definition etc., and the e-edges bind together the beginning with the end of some complex increment as begin-end, rec-end. Finally, the edges which are drawn without label express the order of some constructs which is also the order in which they are displayed on the screen. Some edges correspond to rules of the context sensitive syntax, e.g. the eq-edge expresses that the type of the variable MAN is that given in the type declaration above. The PERSON-node is kept twice in the module graph to have a simple one-to-one correspondence between module graph and source text on the screen. Further edges, which are not drawn in fig. 2, are necessary e.g. for simply handling cursor movement but also

for other technical reasons arising in the context of evaluation and execution of the module graph.

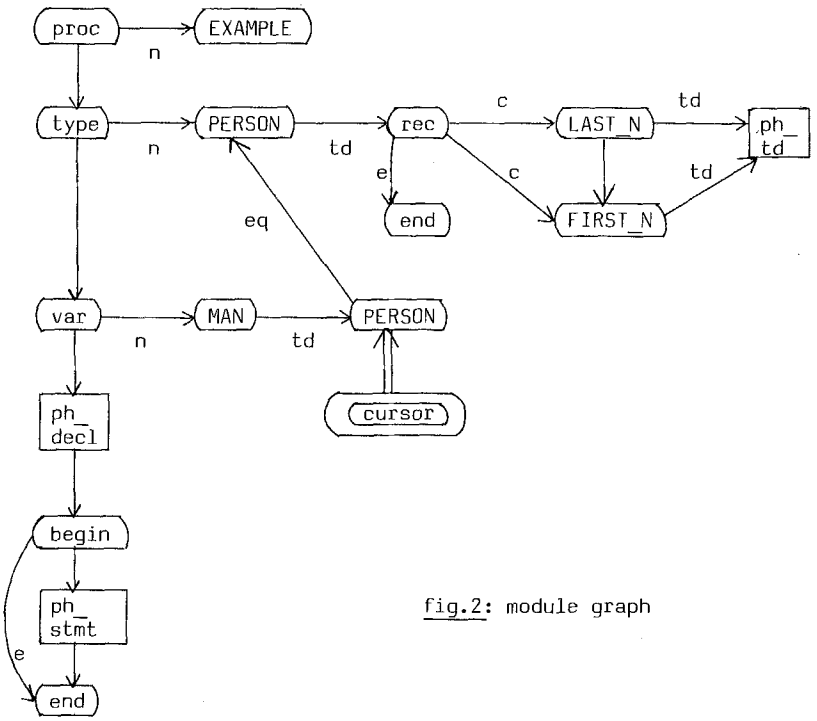


fig.2: module graph

The *translation scheme* pursued in IPSEN is given in fig. 3. Corresponding to the input of editing commands the module graph is altered appropriately. The source code displayed on the screen is generated from the module graph, i.e. the source text is not kept in storage too. This module graph may now be evaluated to find out, if the (partial) program has some property, transformed, or executed. This execution may also happen after having instrumented the module graph by some consumption counters, or this execution may only go on if some test conditions hold true. Also, execution may take place only after having translated the module graph to some other more machine adequate level (incremental compiling). All these aspects of further activities around the module graph are *not* studied in this paper (cf. e.g. /Na 80/).

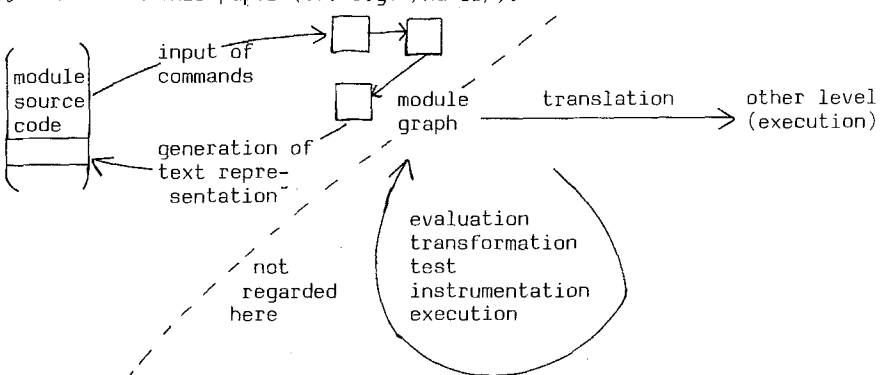


fig.3: translation scheme for Programming in the Small within IPSEN

It is clear that the contents of the screen (of working and command area) as well as the shape of the module graph heavily *depend on the programming language*

to be supported, which in our case is PASCAL. Any admissible modifying command leads to a modification of the module graph and, thereby, also of the source code shown at the screen. In the next section we will show that these module graph changes can easily be described by graph grammars.

4. CONSTRUCTION OF A PROGRAMMED GRAPH GRAMMAR

A *sequential programmed graph grammar* consists of a start graph, a set of productions, and a set of control procedures which control more complicated graph rewritings. A production consists of a left-hand side (the graph to be replaced), a right-hand side (the graph to be inserted) and an embedding transformation (which says, how incoming and outgoing edges are transferred, when the right-hand side replaces the left-hand side). For the embedding tansformations needed in this paper the notation of nearly any graph grammar approach can be used (cf. /CER 79/). A control procedure is nothing else than a flow diagram the actions of which are applications of productions or calls of other control procedures. These control procedures are denoted here in a PASCAL-like fashion in order to make use of control structures. A direct sequential programmed derivation step from graph g to graph g´ by control sequential programmed derivation step from graph g to graph g´ by control procedure c_i, which is abbriviated by

$$g \xrightarrow[c_i]{sp} g',$$

is nothing else than a sequence of elementary sequential derivations with productions p_j which are named by a control path through c_i and all the control procedures called within this path. A sequential programmed derivation then consists of a sequence of such direct sequential programmed derivation steps. The aim of introducing control procedures is to describe modifications of a graph which are the result of a sequence of simple steps rather than the result of a single step.

We will show now that we can construct in a *systematic way* a programmed graph grammar for describing the module graph modifications due to syntax-aided editing *using the following inputs*: (1) the context free grammar of the underlying programming language (here given by PASCAL syntax diagrams (cf. /JW 78/), (2) the context sensitive rules of the language, and (3) the idea of the user interface we have outlined in section 2. This construction is done in three steps: (i) the context free grammar is modified such that it reflects the user interface as given above by menues and frames. Modification here means that some syntax diagrams are made more hierarchical inasmuch as some part of the syntax diagram is taken out and made to another new syntax diagram. On the other hand, the grammar is also flattened by "inline-inserting" syntax diagrams into other ones. (ii) From these modified syntax diagrams the control procedures of the programmed graph grammar can be derived nearly automatically. (iii) The last step is then the development of the corresponding graph productions.

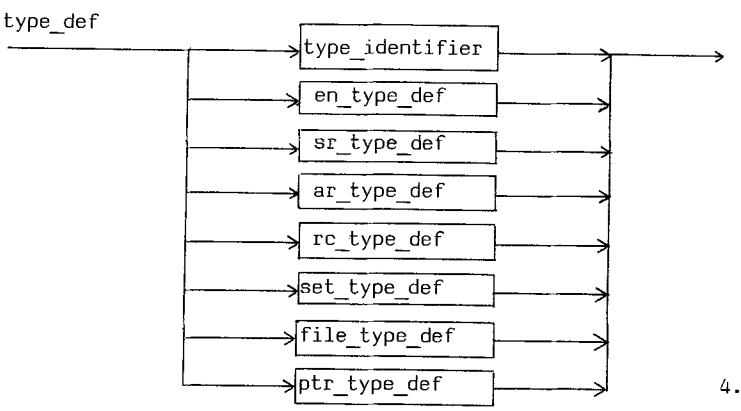
After modifying the context free string grammar we have three kinds of syntax diagrams and, therefore, also *three kinds of nonterminals* : (a) menue type nonterminals, the corresponding syntax diagram is only showing all possible alternatives for this nonterminal one of which has to be selected, but not their internal structure, (b) frame type nonterminals which correspond to complex increments, where, however, only those simple increments and concrete syntax symbols are contained in the syntax diagram which correspond to the frame and, finally, (c) nonterminals corresponding to simple increments.

The *distinction* between *simple* increments and *complex increments* we have made here is rather artificial from the programming language point of view, it is reasonable only from the user´s interface point of view: A simple increment is something which is put in as text string and not by further commands and their parameters. A simple increment may for example be an expression at the right-hand side of an assignment, or the left-hand side of an assignment. This simple increment is simple at the user interface but not at the module graph level. There, of course, the expression is subdivided into some kind of tree structure

and additional edges which are necessary e.g. to check for the context sensitive syntax.

We should make precise what we mean by an *increment*. It is a piece of text derived from a nonterminal of the string grammar, i.e. from the right-hand side of its syntax diagram. This increment may be *empty* (the nonterminals of the right-hand side have not been replaced), *partially expanded* (some nonterminals may be replaced), or *totally expanded* (there are no nonterminals left). For any increment there exists a subgraph of the module graph, i.e. for any text increment there exists a graph increment.
For any increment there also exists a *string representation* on the screen (as part of the "source code") where nonterminals are not displayed. To complex increments there corresponds a frame which is a part of this string representation (with additional comments). It contains only the simple increments of the uppermost level of the increment. If the complex increment is empty then, of course, the frame is also empty. Whereas frames to complex increments have some structure, those belonging to simple increments have none, i.e. the frame is a single input field. To make this distinction clear we speak of *gaps* (as unstructured frames to simple increments) and not of frames.

Let us now demonstrate, how the control diagrams of the programmed graph grammar can be derived from the syntax diagrams of the modified (context free) string grammar of the underlying programming language. The first type of syntax diagrams where we make clear how to proceed is the *menue type*. Fig. 4.a shows the syntax diagram for type definition. This syntax diagram directly corresponds to the menue of fig. 1.c. The modification of the original PASCAL syntax diagram (cf./JW 78/) here consists of the introduction of an additional level of hierarchy: Usually the type definitions for enumeration type up to pointer type are directly contained within the syntax diagram for type definition. The translation of the syntax diagram of fig. 4.a into the control procedure of fig. 4.b is trivial. Please note that for this kind of syntax diagrams no graph productions have to be developed as the modification of the module graph is only done in the control procedures called by the control procedure type definition. Thus, the function of this kind of control procedures is nothing but calling the control procedure corresponding to the selection the user has made.



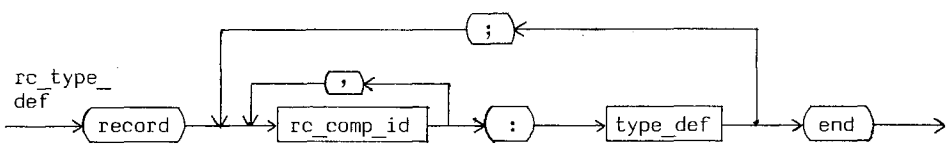
4.a

The next type of syntax diagrams to be discussed is the *frame type*. Fig. 5.a shows the original syntax diagram for rc_type_def of the PASCAL syntax (variant records are not regarded here) and fig. 5.b the modified syntax diagrams due to the intuition of frames sketched above in section 2. As the component type definitions cannot be contained within the frame for rc_type_def we have removed the whole record component and made another syntax diagram out of it. Now, the frame to rc_type_def in fig. 5.b only consists of the two word symbols record and end, i.e. it has no input field. This is why we have transferred this frame immediately into the working area without displaying it in the command area (cf.

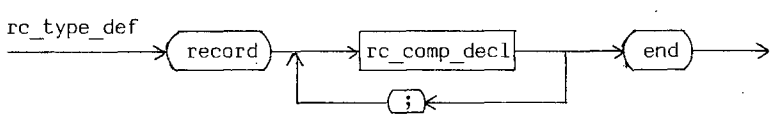
fig. 1.d). This example is not typical for a frame type syntax diagram insofar as this frame contains no gaps for simple increments. More typical is for example the for-statement syntax diagram with gaps for loop variable, upper and lower bound which we explain below.

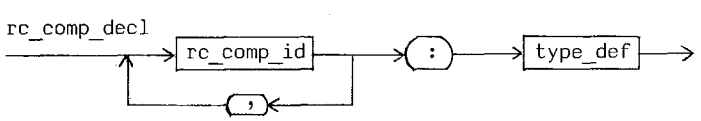
```
control proc type_def;
begin
   case "user choice" of  -- by input of a cmd name or cmd condition
      IT:   type_identifier;
      IENT: en_type_def;
      ISRT: sr_type_def;
      IART: ar_type_def;
      IRT:  rc_type_def;                                    4.b
      IST:  set_type_def;
      IFT:  file_type_def;
      IPT:  ptr_type_def;
      otherwise: ERROR( cmd area: "Only type definition allowed.")
   esac
end;
```

fig.4: menue type syntax diagram and corresponding control procedure



5.a





5.b

```
control proc rc_type_def;
begin
   rc_skeleton;   -- frame is immediately transferred
   while "another record component" do  -- by command IRC
      rc_comp_decl
   od;
   erase_ph_cl_if_cl_nonempty
end;

control proc rc_comp_decl;
begin
   rc_comp_skeleton;
   while "input of a new component identifier" do
      rc_comp_id   -- input is taken; context sensitive check
   od;                                                      5.c
   erase_ph_idl_if_idl-nonempty
   -- frame is closed and transferred; control back to user;
   -- insertion of type definition within procs called in type_def.
end;
```

fig. 5: frame type syntax diagrams and corresponding control procedures

The translation into a control procedure is straightforward. The structure of the modified syntax diagrams of fig. 5.b can directly be found within the procedures of fig. 5.c. At the beginning of each control procedure, however, there is an application of a skeleton production in addition which inserts the concrete syntax nodes and some placeholder nodes in the module graph as we show in detail below. At the end of each control procedure we find the application of some technical productions which delete some placeholder nodes which are not necessary further. The function of the control procedure rc_type_def is - besides of applying a technical production - to call the control procedure rc_comp_decl as often as the user indicates. The frame for record component declaration contains a list of identifiers but not the corresponding type definition. For each identifier put in by the user some context sensitive check is necessary here to avoid that this identifier has already been used as component identifier within the actual record type definition. The insertion of the subgraph corresponding to the type definition of any component is done within the procedures which are called within the control procedure type def.

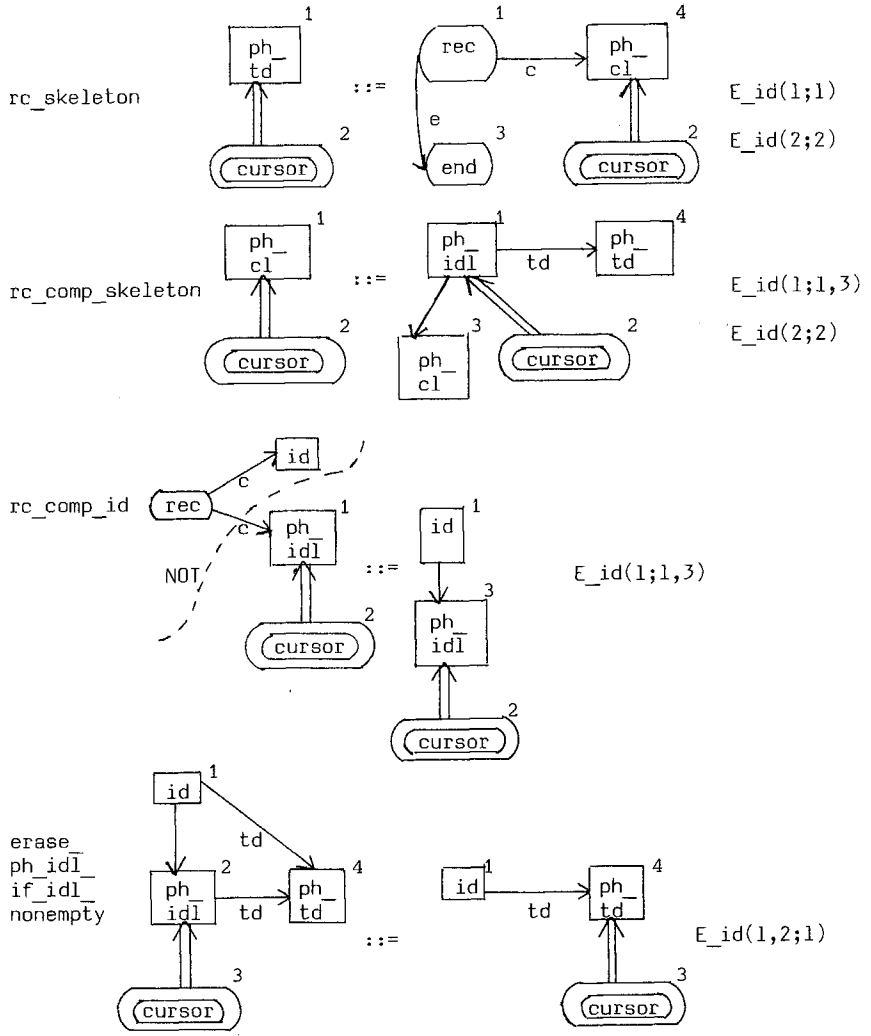


fig.6: graph productions of control procedures rc_type_def, rc_comp_decl

Let us now explain how the *productions* for the two control procedures rc_type_def and rc_comp_decl look like (cf. fig. 6). The production rc_skeleton

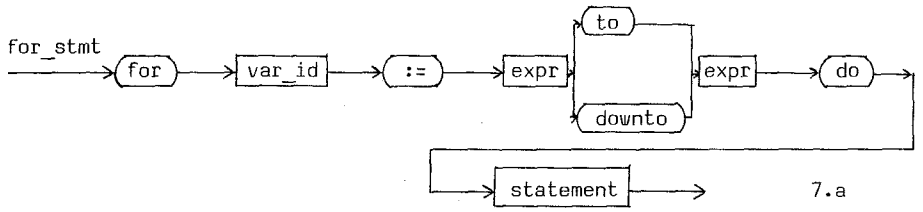
inserts a pair of rec-end-nodes, but also changes the placeholder node from ph_td to ph_cl where cl stands for record component list. The cursor is moved to the ph_cl-node. The embedding transformation is such that all edges of node 1 of the left-hand side are transferred without any change to node 1 of the right-hand side and the same happens for edges incident to node 2 of the left- and right-hand side, respectively. This is indicated by E_id(1;1) and E_id(2;2). Analogously, the production rc_comp_skeleton inserts two further placeholder nodes, namely for identifier list and for type definition. The ph_cl-node for record component list is still existing. The cursor now is at the ph_idl-node, as identifiers for record components are expected. The cursor-node gets again an identical embedding while the embedding of the node 1 of the left-hand side is now transferred to node 1 *and* 3 of the right-hand side. This means both that the ph_idl-node as well as the ph_cl-node of the right-hand side have an incoming c-edge after the application of this production. The production rc_comp_id inserts a record component identifier leaving the ph_idl-node available as further identifiers are expected. Note, however, that this identifier node is only inserted if within the same record type definition there is no record component with the same name. This is expressed by the negative application condition drawn here as a subgraph separated from the left-hand side by a dotted line marked by NOT. The label id within this production stands for an arbitrary identifier. So, we furthermore have some primitive two-level mechanism here. Finally, the technical production erase_ph_idl_if_idl_nonempty erases the ph_idl-node if and only if there is already some identifier node. If there is none, then the placeholder node is preserved, as any record component declaration must have at least one identifier. The reason that we erase the placeholder node is that in the full incremental mode (see below) nearly everywhere a change can occur and, because of storage and lucidity reasons, we cannot insert everywhere a placeholder node. So, to act uniformly, the placeholder node is also deleted here. The embedding transformation of this production means that all edges bordering node 1 and 2 of the left-hand side are edges of node 1 of the right-hand side after application of the production. The technical production erase_ph_cl_if_cl_nonempty is completely analogous and, therefore, not given here.

To summarize the *graph rewriting approach* used in this paper here we can state: the embedding transformations are rather simple. No relabelling or reversing of embedding edges is necessary. We furthermore need some primitive two-level mechanism, as identifiers put in by the user must replace metasymbols of node labels thereby producing so-called productive productions. Finally, we make use of negative application conditions. The graph grammar presentation of this paper is completely informal, for precise definitions see /Na 79/.

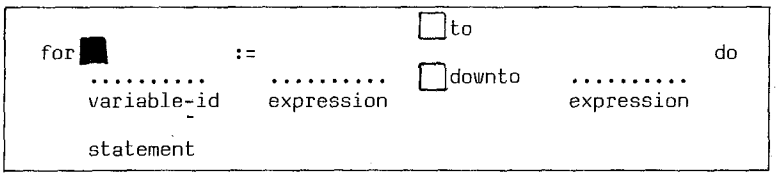
As another example of a frame type nonterminal and its translation into a control procedure look at fig. 7. There, fig. 7.a gives a syntax diagram which again is taken out of a syntax diagram of PASCAL to fit the frame intuition we gave above. In fig. 7.b the corresponding frame at the screen is shown and fig. 7.c gives the control procedure. The productions are analogous to the example above and, therefore, are not given here. This example will be picked up again in section 6.

For *simple increment* nonterminals we give no example in this paper. If a simple increment is only a node label on the module graph level, then the simple increment is only a node label on the module graph level, then the control procedure is only the application of a trivial relabelling production (eventually together with a context sensitive check). If, however, a simple increment is internally represented as a graph rather than a single node, then this graph has to be built up and embedded in the module graph. Then, also a lot of context sensitive checks are necessary, i.e. for making sure that all applied occurrences belong to declared objects, types, procedures etc. This modification of the module graph due to the input of a simple increment can also be described by programmmed graph grammars in an analogous proceeding as sketched above for frame type nonterminals. Here, again, the guideline for the construction of the programmed graph grammar is the context free grammar which, however, in this part is not modified as these increments are regarded to be simple at the user

interface.



7.a



7.b

```
control proc for_stmt;
begin
   for_stmt_skeleton;    -- loop upward as default                    7.c
   var_id;               -- includes context sensitive check
   expression;           -- c.s. check; construction of an internal graph
   if "downto" then change_to_downloop;
   expression;           -- c.s. check; construction of an internal graph
   -- frame is closed and transferred; control back to user;
   -- insertion of stmt graph within control procs called within statement
   statement
end;
```

fig.7: another frame type nonterminal: syntax diagram, frame, control procedure

5. THE FULL INCREMENTAL EDITING MODE: USER INTERFACE REVISED

In section 2 we have sketched syntax-aided editing for the *input mode*. Menues and frames are offered to the user to select syntactical constructs and to put in their simple increments. All necessary syntactical checks are carried out and, on the other hand, the concrete syntax is generated by the system rather than put in by the user. The cursor is set forward automatically. The building-up of the module graph has been specified within control procedures which recursively call each other. The user only is asked to select between alternatives possible in a special situation. Now, in the *full* incremental editing *mode* there is no sequential and fixed order in which editing commands are put in by the user. Any order of inserting, changing, deleting, or cursor moving commands is possible.

To illustrate this, let us again consider a *dialog fragment* (cf. fig. 8). In this example we regard the text input mode for commands, i.e. the commands are chosen here by putting in text strings for command names. In fig. 8.a the working cursor is before an if-then-else-statement, which we want to refine partially. Pressing three times the next-button positions the cursor to the location, where a boolean expression is to be put in. (Pressing it once the block, if twice the if-then-else-statement is marked).
Now, within the command area we put in the command IBE for Insert Boolean Expression (cf. fig. 8.b). (I for Insert in this case would have been enough, as at this position only a boolean expression is allowed.)
As above, a frame appears which, however, is unstructured here as we regard a boolean expression to be a simple increment. After putting the string A>B into tne input field of the frame and pressing the next-button the working cursor is

at the position of the then-part.
This then-part shall be left empty for a while. So, by pressing again the next-button, we move the cursor down to the else-part. Then we put in the command name IAS for insert assignment statement.
Here, a structured frame appears, wich contains the becomes symbol. The input sequence A (for the variable at the left-hand side), next-button for moving the command cursor to the right-hand side) and 1 (for the right-hand side) completes the assignment. The following next-command moves the cursor to the next position, which here is the following assignment.
Here, for example, the command DAS for delete assignment statement would delete this statement. The dialog could proceed anyhow now.
What we can learn for an arbitrary editing step is that an arbitrary increment can be partial (empty, partially expanded, totally expanded) before being incrementally edited but it may be partial even after editing. For example, in fig. 8.b the if-then-else statement is completely empty, afterwards in fig. 8.f its then-part is still missing.

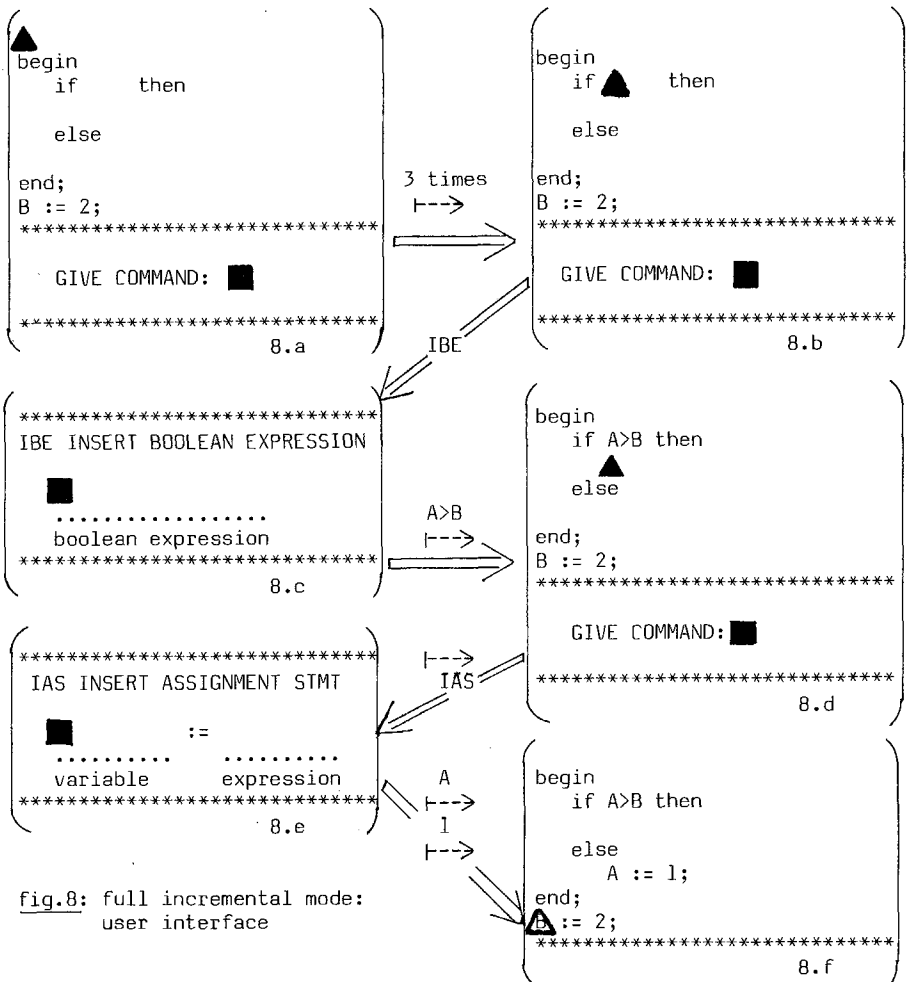


fig.8: full incremental mode: user interface

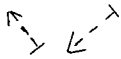
Now, let us explain, which *commands* are possible in the full incremental editing mode.
There are *insert* commands which can be used to fill an existing gap for a simple increment (e.g. for boolean expression if the enclosing if-then-statement is already generated) or to generate a gap and possibly fill it (e.g. for inserting

an assignment statement). Analogously, a complex increment can be inserted and its frame can be filled (left blank, be partially filled, be completely filled). Inserting a complex increment means also the insertion of concrete syntax nodes and placeholder nodes. Finally, a partially or totally expanded increment may be inserted which is the result of some previous dialog activity (see below) which means that some graph has to be embedded in the module graph.
As in most situations there are several possibilities for expansion, insert is not a command but a *command group*. IAS or IBE are commands. However, in some situations the command is clear from the context and, therefore, only the command group has to be specified. This remark also holds true for the following command groups.
Delete commands are possible for simple increments and complex increments. If the increment to be deleted is obligatory (e.g. the boolean expression within an if-then-statement) then in the module graph a placeholder node is left behind after having erased the subgraph corresponding to the increment. Otherwise, if the increment is optional the increment subgraph is completely erased within the module graph. Please note, that for complex increments this means that all inner increments are also deleted.
For making incremental modifications easier there are also *change* commands which avoid (a repetitive) deletion and insertion of simple increments whithin an increment. If, for example, the command CFS (for change for-statement) is given, then the frame for for-statement again appears and all simple increments in the frame can be changed (whithout touching all possible inner increments of the for-statement).
Finally, there often arises a situation that a complex increment has to be transformed to another one, e.g. the transformation of an if-then-statement into an if-then-else-statement, of a block into a procedure body etc.. As there are many situations feasible and reasonable a big bunch of commands would result if for any of these transformations there would exist a corresponding command. For this, there are *save* commands with which an increment or a sequence of increments can be saved to be used later. This means that a more or less big part of the module graph must be stored such that it can be inserted at any admissible position later only by specifying some name (which is asked for when executing the saving command).

For moving around arbitrarily we must also have *cursor movement* commands (cf. table 9).
One of them is the *next*-command which is initialized by pressing the right arrow button. In section 2 this button was understood as the end symbol of an insertion command. Now, in the full incremental mode it is a command like all other commands which is only activated differently, namely by pressing a special key. "Next" means moving the cursor to the next increment if we follow the most detailed source structure. This sometimes means to go into a structure (from if-then-statement to the boolean expression within the if-then-statement), to go to the next structure on the same level (from the boolean expression to the then-part of an if-then-statement), but also to go to the next structure at a higher level (from the then-part to the increment following the if-then-statement). The *pred* -command (pred for predecessor) initiated by pressing the left arrow key is inverse to the next command, i.e. it is going up within the most detailed source structure.
The *go-up* -command (↖ key) takes the cursor up to the beginning of the next increment upward in nesting hierarchy, the *leave*-command (↙-key) exits the actual increment and then goes to the beginning of the next following increment, if any, otherwise again up and forward. Finally, the ↧ and ↥ cursor movements have been introduced for going *down* and *up* without entering the details of an increment. Thus, pressing the ↧-key if the working cursor is at an if-then-statement means that the incremment following the if-then-statement on the same (or next higher level) is marked.
It is clear that these cursor movement commands can easily be implemented on the graph grammar level by writing the corresponding control procedures and their elemetary cursor movement productions.

command "next" and "pred": to next or predecessing increment in the most detailed structure (in graph and source text)

command "down" and "up": to following increment or to increment heading of the actual increment without entering the details of the actual or heading increment



command "hierarchy up" and "leave": go up in nesting hierarchy, or leave actual increment and then take next (eventually again leave and next increment)

table 9: cursor movement commands

The *input mode*, which we have introduced in section 2, is only a *special case of the full incremental mode*, i.e. it is only some abbreviation. One step in this direction of interpreting the input mode in this way was to understand the pressing of the ⊢--> key always as some *movement* command. The next is to regard a frame as part of the source: A frame is nothing else than a *cutout* of the program which is enriched with comments. It can be filled but also left by cursor movement commands. The third step, finally, is to understand the filling of input fields as *implicit* input of an *insert* command together with its parameter. The possible command is clear within such a situation. So, in fig. 10 the input LOOPV is understood as implicit activation of a command IV for insert variable identifier with parameter LOOPV, ⊢--> as movement command to the next placeholder node, 1 as implicit activation of IEX (insert expression) with parameter 1. Finally, we can see from this example that within a frame also some selection can occur and that a frame can also be left by a command, here leaving the expression for the upper bound blank.

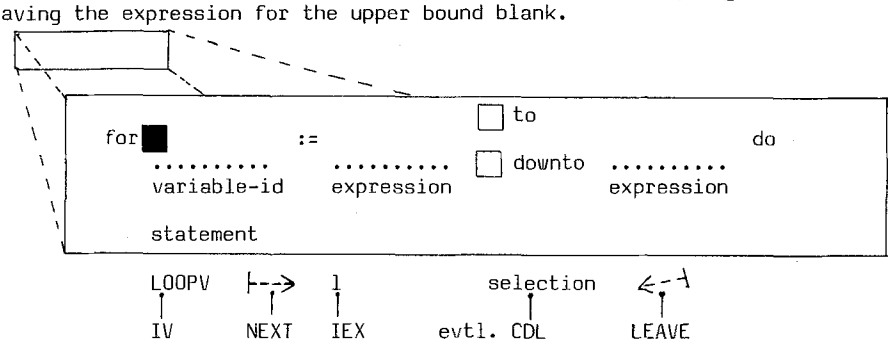


fig.10: input mode as special case of the full incremental editing mode

6. CONTROL PROCEDURES REVISED

What was the *execution model* for sequential programmed rewriting steps we had for the *input mode* control procedures of section 3? There, the control procedures have been activated by (recursive) calls. The order of activation was fixed within the bodies of the control procedures. For example, in the control procedure for_stmt of fig. 11 it was fixed that after applying a skeleton production the control procedures var_id, expression, expression, and statement are called in this order. The user was only asked, if one of more alternatives had to be selected. The cursor movement in the graph as well as on the screen was understood to happen automatically.

In the full incremental mode no predetermined and automatic activation of control procedures can take place. The reason is that the user is allowed to put in increments in any order, leave partial increments, come back to partial increments, delete increments etc.. Here, all control procedures are *directly activated* in any order by the user rather than by some kind of pregiven order fixed in the bodies of the control procedures. This direct activation can be

done *explicitly* by specifying a command (by input of a command name or by a selection) or *implicitly* by filling out the input fields of a frame.

So, if we write the control procedure for a for-statement in the full incremental mode (cf. fig. 11), then this control procedure need not contain the activation of var_id, expression and statement as these control procedures are directly activated. Also, the cursor movement need not be contained nor done automatically. Finally, the change from an upward-loop to a downward-loop is directly activated by a corresponding change command. So, the control procedures in the full incremental mode (cf. fig. 11.b) consist only of a *skeleton production* for complex increments or some *relabelling production* for simple increments which may contain context sensitive checks. However, because of the arbitrary order of activation, at the beginning of each control procedure for the full mode, there must be a *check* whether the execution of this control procedure is allowed at all. This check need not be carried out, if the command is selected in menue mode and it is also not necessary for directly but implicitly called insert commands.

```
control proc for_stmt;          -- corresponding to   IFS
begin                                                  |
   for_stmt_skeleton;           -- ____________________|
   var_id;                      -- corresponding to   IVI |--->
   expression;                  -- corresponding to   IEX |--->
   if "downto" then change-to-downloop; -- c.t.       CDL
   expression;                  -- corresponding to   IEX |--->
   statement                                                    11.a
end;


control proc I_For_Statement;
begin
   exit if not for_stmt_allowed; -- check only for non-menue mode
   for_stmt_skeleton             -- contains implicit NEXT call
end;                             -- show frame
                                                                11.b

control proc I_Var_Id;
begin
   exit if not var_id_allowed;   -- check not necessary if impl. activ.
   var_id                        -- corr. frame only shown if expl. activ.
end;                             -- context sensisitive check
```

fig.11: control procedures for inserting a for-statement: input mode and full mode

Now, again, let us compare the *execution model* we had for *input mode* in section 3 with that for the *full mode* we have here (cf. fig. 12). For input mode we had one rather complex programmed graph rewriting step which was driven by the execution of a single control procedure c_prog_im (for program and input mode) corresponding to a PASCAL module (program or subprogram). The mutual activation of control procedures was already fixed in the bodies of the control procedure c_prog_im and its subordinate control procedures which recursively called each other. User input was only necessary for selection and input of simple increments. This complex rewriting step directly corresponds to a derivation of the source program within the PASCAL string grammar. The graph grammar for input mode is nothing else than a rather direct translation of the corresponding string grammar.
In the full mode we have no correspondence to a string derivation as the module and also the internal graph is usually partial before and afterwards. Furthermore, it can be changed arbitrarily. So, the situation of the full mode is that we have a sequence of sequential programmed derivation steps with control procedures c_i_j selected by the user. If such a control procedure is not admissible in a special situation, then its execution is rejected because of the check for applicability at the beginning of each control procedure.

Any of these graphs g_i_j of the full mode graph grammar is also the result of a derivation of the input mode graph grammar. The application of a programmed step corresponding to full mode on graph g_i_j and leading to graph g_i_j+1 can be imagined as changing the derivation of g_i_j within the input grammar in order to get a derivation of g_i_j+1 within the input grammar. Furthermore, it is clear that the input grammar is properly contained in the full mode grammar.

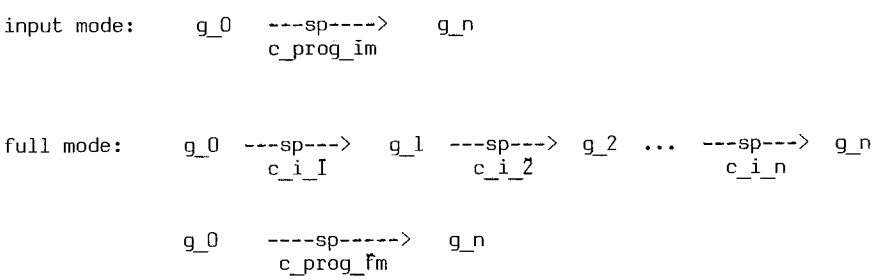


fig. 12: programmed derivations in input and full mode

Now, we summarize this sequence of sequential programmed rewriting steps corresponding to a user session of full mode editing in order to get a complex step with one control procedure. This control procedure c_prog_fm (for program and full mode), of course, then has the structure of a while-loop (cf. fig. 13). As long as commands are put in by the user, the while-loop is executed. Depending on the command a corresponding control procedure is called. Commands can be input commands, delete commands, change commands, save commands, cursor movement commands, but also other commands arising in the context of evaluation, transformation, execution, testing, and monitoring of modules.
The module graph acts as a global data structure for all control procedures. The calling hierarchy of the recursively called control procedures of the input mode is implicitely contained in the module graph.

```
control proc prog_fm;
begin
   while "command given" do
      case "command" of                          -- commands:
         CMD1: call_of_control_proc_to_CMD1;     --          I...
          .                                      --          D...
          .                                      --          C...
          .                                      --          S...
         CMDn: call_of_control_proc_to_CMDn      --          cursor, etc.
      esac
   od
end;
```

fig.13: uppermost control procedure for full incremental mode

Now, if we summarize the proceeding taken in this paper, we get the picture of fig. 14. We have seen that we can *systematically develop a programmed graph grammar for syntax-aided editing* . The input of this proceeding is a clear idea of the user interface, and the context free as well as the context sensitive syntax of the underlying programming language. The user interface leads to a modification of the context free string grammar, thus influencing the programmed graph grammar indirectly. On the other hand, we have also a direct influence as in the control procedures we specify transfer of frames, area for error messages etc. too. This systematic development is applicable for the input mode as well as for the full editing mode. Furthermore, we have seen that the input mode is only a special case of the full mode.

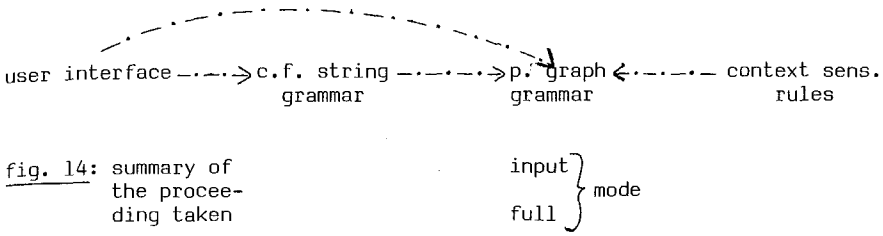


fig. 14: summary of the proceeding taken

7. PROGRAMMING IN THE LARGE

We claim that for Programming in the Large we can take the same systematic proceeding which was described for Programming in the Small above. However, the *starting point* is quite *different* here. For Programming in the Small the base of support is the underlying programming language. No method of using this programming language is supported up to now as , for example, stepwise refinement. So, making reasonable or foolish use of PASCAL is not influenced by IPSEN.
For Programming in the Large, i.e. for specification purposes, we cannot take the same view. Old programming languages as FORTRAN, or COBOL, but also newer programming languages like PASCAL hardly offer any constructs evidently applicable for Programming in the Large. So, here some formal language and some methodology for developing and maintaining specifications has to be offered. This means, that some *module concept* has to be selected and the development of specifications using this module concept has to be facilitated. For this, we again make use of *all* IPSEN characteristics (incremental mode, syntax-directed reaction, command-driven input etc.). Also, the transformation of such specifications into an existing programming language has to be supported. The necessity for a methodological support even arises for a quite modern programming language like ADA. In ADA, there are a lot of constructs applicable for Programming in the Large, i.e. ADA can be used as specification language. We feel, however, that their methodological use should be facilitated in order to get elucid specifications. The reader may have noticed that we mean only syntactical aspects here if we speak of specification.

To speak of a module concept especially means to classify certain necessary *types* for *modules* . In IPSEN, we have chosen different types for data abstraction and for functional abstraction, respectively. Furthermore, some *relations between modules* have to be fixed. It is our belief that for this at least the following relations are necessary: "A module B is contained in a subsystem A´ (represented by the top module A of the system) and therefore is usable in some local context", and "A subsystem A´ represented by its top module A is usable as some common tool by other subsystems". In both cases a module exports resources which have to be imported explicitly by other modules. Besides module types and module relations a module concept consists also of a set of *consistency conditions* (part of the context sensitive syntax).
Looking at existing programming languages then this module concept represents some kind of extension to these programming languages, i.e. we must introduce a textual representation for these new constructs. It is, however, not an extension in the sense that a precompiler is planned to be written as, in the same way as in Programming in the Small, the source code on the screen is generated from an higher level intermediate data structure, namely the system graph. There is no place left to go into details of this module concept here. The reader is referred to /Ga 82a,b/ and a forthcoming paper.

Having fixed the module concept the next step is to lay down its *representation* as a *graph*, i.e. the class of graphs used as system graphs: node labels, edge labels, graph consistency conditions. The following step then is to fix the *increments* for incremental changes. Trivially, in our approach an increment is not the source of a complete module as it is the case in those languages which

have constructs for separate compilation (as ADA, cf. e.g. /Na 82/). Instead increments are, for example, parts of the module interface , i.e. of the export or import clause. After having determined the graph representation and the kind of increments the *abstract syntax* on graph level is roughly fixed. So, up to this point, we have some informal specification of the corresponding graph grammar. This belongs to context free as well as context sensitive aspects.

The next step consists of fixing the *string representation* for this specification language derived from the module concept. This is a formal programming language for Programming in the Large but, as stated above, no compilation step in the classical sense necessarily appears. This fixing is done in the form of syntax diagrams.

Now, the proceeding of sections 2-6 of above can be adopted as all three inputs for our proceeding are laid down. The derivation of the sequential programmed graph grammar may of course lead to modifications of the informal graph grammar specification mentioned before.

8. CONCLUSION

We have shown that graph grammars are an appropriate specification mechanism for incremental changes arising in the context of syntax-aided Programming in the Small and Programming in the Large, respectively. The specification was carried out in a systematical or engineering-like way: It was the result of a rather *mechanical transformation* using three inputs. The proceeding was first demonstrated for the input mode and then extended to the full incremental mode of Programming in the Small. Finally, we have sketched that it can be used also for syntax-aided editing within Programming in the Large.

As stated in the introduction, the graph grammar specification has a two-fold significance. On one side it makes *precise* which kind of problems occur and how an abstract solution to these problems looks like. On the other hand this specification is operational and, therefore, is a direct guideline for the *specification* of IPSEN in the software-engineering sense.

What we pointed out is rather the *method* taken than its result. While the result is depending on the programming language for Programming in the Small and the module concept for Programming in the Large the proceeding, of course, is also applicable for other programming languages and module concepts. Moreover, we would claim that this proceeding can be applied for arbitrary *dialog systems*. Especially, it is also applicable for the third problem area "organizational items" within IPSEN. Because of this general suitability we have chosen the more general title of this paper.

REFERENCES

/BN 82/ Burkhart, H./ Nievergelt, J.: Structure-oriented editors, Informatik-Fachberichte 30, pp. 164-184, Berlin: Springer-Verlag

/Bu 80/ Buxton, J.N.: Requirements for the ADA Programming Support Environment ("Stoneman"), United States Department of Defense

/CER 79/ Claus, V./ Ehrig, H./ Rozenberg, G. (Eds.): Proceedings of the International Workshop on " Graph Grammars and their Application to Computer Science and Biology", Lecture Notes in Computer Science 73, Berlin: Springer-Verlag

/DG 80/ Donzeau-Gouge, M. et.al.: Programming Environments Based on Structured Editors - The MENTOR Experience, Techn. Report 26, INRIA, France

/ES 82/ Engels, G./ Schäfer, W.: Specification of a Programming Support Environment by Graph Grammars, to appear in the Proceedings of the WG '82 on "Graphtheoretic Concepts in Computer Science", Munich: Hanser Verlag

/Ga 82a/ Gall, R.: Structured Development of Modular Software Systems: The Module Graph as Central Data Structure, in Proceedings of the WG '81 on "Graphtheoretic Concepts in Computer Science", Munich: Hanser-Verlag

/Ga 82b/ Gall, R.: Dissertation, to appear

/Ha 82/ Habermann, N. et.al.: A Compendium of GANDALF Documentation, Techn. Report, May 1982, Department of Computer Science Carnegie-Mellon University, Pittsburgh

/JW 78/ Jensen, K./ Wirth, N.: PASCAL User Manual and Report, sec. ed., New York: Springer-Verlag

/Me 82/ Medina-Mora, R.: Syntax-directed editing - Towards Integrated Programming Environments, Techn. Report CMU-CS-82-113, Department of Computer Science, Carnegie-Mellon University, Pittsburgh

/Na 79/ Nagl, M.: Graph-Grammatiken - Theorie, Anwendungen, Implementierung, Wiesbaden: Vieweg-Verlag

/Na 80/ Nagl, M.: An Incremental Compiler as Component of a System for Software Development, Informatik-Fachberichte 25, pp.29-44, Berlin: Springer-Verlag

/Na 82/ Nagl, M.: Einführung in die Programmiersprache ADA, Wiesbaden: Vieweg-Verlag

/Sch 75/ Schneider, H.J.: Syntax-oriented Description of Incremental Compilers, Lecture Notes in Computer Science 26, pp.192-201, Berlin: Springer-Verlag

/TR 81/ Teitelbaum, T./Reps, T.: The Cornell Program Synthesizer - A syntax-directed Programming Environment, Communications of the ACM, Vol. 24, No. 9, pp. 563-573